ADVANCED RUST PROGRAMMING

FROM MASTERY TO EXPERT SYSTEMS CRAFT

Go beyond the basics. Master Rust's deep internals and build high-performance, reliable systems.

- MEMORY LAYOUT & VARIANCE
- TRAIT RESOLUTION & COHERENCE
- ATOMIC OPERATIONS & MEMORY MODEL
- ASYNC INTERNALS & EXECUTOR DESIGN
- UNSAFE CODE & SAFETY CONTRACTS
- FFI BOUNDARIES & INTEROP
- ALLOCATOR DESIGN & MEMORY MANAGEMENT
- PERFORMANCE ENGINEERING

JAMES REDFORD

Advanced Rust Programming

From Mastery to Expert Systems Craft

Steve Publications

This book is available at https://leanpub.com/advancedrustprogramming

This version was published on 2026-08-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

From Mastery to Expert Systems Craft

This book takes experienced Rust developers beyond ownership, borrowing, and basic concurrency into the deep machinery of the language — variance, memory layout, trait resolution, atomic operations, async internals, unsafe contracts, FFI boundaries, allocator design, and performance engineering. Each chapter builds precise mental models with complete, compilable examples, explaining not only how features work but why they exist, when to use them, what trade-offs they entail, and how they compose into production-grade systems.

Introduction: The Expert's Rust

There is a moment every Rust developer eventually encounters. You have shipped services that never segfault, written concurrent code without data races, and tamed the borrow checker well enough to build real systems. Then someone asks you to write a lock-free queue, or implement a custom allocator for an embedded device, or optimize a hot path where dynamic dispatch is chewing through cache lines, or reason about why your async task leaks memory when cancelled mid-operation.

Suddenly the comfortable patterns feel insufficient. The borrow checker seems to be rejecting code that should be safe. Trait error messages span pages and make no sense. You find yourself reading compiler source code just to understand what the compiler is actually complaining about.

This book exists for that moment. It is not a tutorial on ownership or lifetimes. It assumes you already know how to write Rust and want to understand why it works the way it does, how its pieces fit together, and how to wield them at an expert level. Mastery of Rust is not merely knowing what compiles; it is understanding the mental models that make the language tick so you can reason about correctness, performance, and maintainability in complex systems without guessing.

What This Book Assumes You Already Know

Before we begin, let us be explicit about prerequisites. This book targets developers who have:

- Built non-trivial Rust projects and are comfortable with ownership, borrowing, lifetimes, and the borrow checker's basic behavior
- Used traits and generics to write reusable code, including associated types and trait bounds
- Worked with enums, pattern matching, iterators, closures, and Result/Option for error handling
- Written concurrent code using threads, channels, Mutex, and Arc

- Used Cargo to manage dependencies, build profiles, and run tests
- Encountered the async/await syntax and used at least one async runtime such as Tokio

If you are still wrestling with why a reference does not live long enough or what a trait bound means, this book will be frustrating. Go first through The Rust Programming Language or a comparable resource, then return here when you are ready to go deeper.

Why Go Deeper: The Gap Between Competent and Expert

Competence in Rust means writing code that compiles and runs correctly. Expertise means understanding the guarantees your code provides, knowing exactly where those guarantees come from, and being able to construct new abstractions that maintain safety while achieving performance goals that would be impossible with naive approaches.

Consider a concrete example. A competent developer knows that Rc is not Send and will cause a compile error if moved across threads. An expert understands why: the reference count update in Rc::clone uses non-atomic operations, so concurrent clones from multiple threads would produce data races. The expert also knows that Arc fixes this with atomic operations at a measurable performance cost, and can explain when that cost matters and when it does not.

Or consider dynamic dispatch. A competent developer knows that trait objects use runtime dispatch through a vtable. An expert knows the exact layout of a fat pointer, understands how vtable lookups affect instruction cache behavior, can measure the overhead with criterion.rs, and knows when to replace dynamic dispatch with static dispatch via generics or enums for performance-critical paths.

This gap between competent and expert is where systems programming lives. It is the difference between using Rust as a safer C++ and using it as what it actually is: a language whose type system, memory model, and zero-cost abstraction philosophy compose into something unique. Expert Rust developers do not fight the compiler; they use its constraints as design tools that eliminate entire classes of bugs at compile time while extracting maximum performance from hardware.

The Mental Models That Matter Most

Throughout this book you will encounter several mental models that recur across different topics. Understanding these models is more valuable than memorizing any individual feature:

The ownership model is about aliasing and mutation, not just memory management. Rust's core discipline — mutation xor sharing — prevents data races by construction, but its deeper purpose is to give the compiler information it needs for aggressive optimization. When you understand that &mut T means "I am the only one who can see or change this value right now," you begin to see how this enables both safety and speed.

The type system is a specification language. Traits are not just interfaces; they are contracts that encode invariants, capabilities, and behavioral guarantees. PhantomData marks ownership without storing data. Sealed traits restrict extensibility by design. Type-state patterns make invalid states unrepresentable at compile time. Expert Rust developers use the type system to document and enforce correctness properties that other languages would push to runtime or documentation.

Safety is a property of abstractions, not individual lines of code. Unsafe blocks are not dangerous by themselves; they are regions where you, the programmer, assume responsibility for invariants the compiler cannot verify. A well-designed unsafe abstraction exposes only safe APIs and maintains its invariants internally. The danger lies not in using unsafe but in writing unsound abstractions that allow safe code to trigger undefined behavior.

Performance is a property of your data layout and algorithm, not your language choice. Rust's zero-cost abstractions mean that high-level patterns compile down to efficient machine code when designed correctly. But poor choices — excessive dynamic dispatch, cache-thrashing data structures, unnecessary heap allocations — will hurt performance regardless of language. Expert Rust developers profile before optimizing, understand memory hierarchy effects, and choose abstractions with their runtime cost in mind.

Concurrency requires reasoning about ordering, not just parallelism. Atomics with relaxed ordering are not automatically correct just because they are lock-free. Memory models exist because hardware and compilers reorder operations for performance. Understanding acquire-release semantics, sequential consistency, and the happens-before relationship is essential for

writing correct concurrent code that does not rely on accidental behavior.

How to Read This Book

This book progresses from advanced language concepts through expert-level systems programming. The chapters build on each other: understanding variance requires grasping ownership deeply; understanding async Rust requires knowing about Pin and self-referential types; understanding unsafe abstractions requires knowledge of aliasing rules and memory layout.

Each chapter is organized into focused sections with complete, compilable code examples. Do not skip the examples — they are carefully designed to demonstrate concepts that would be difficult to convey in prose alone. Many examples show both correct and incorrect approaches so you can see exactly where and why things go wrong.

You do not need to read this book linearly if you already have specific interests, but expect some friction when jumping ahead. The chapters on ownership and traits provide foundations used throughout the rest of the book. The chapters on concurrency build on each other. The async chapter assumes familiarity with Pin from earlier material.

Where topics are genuinely unresolved or subject to ongoing language evolution — specialization, negative impls, async drop, scoped tasks — I will be explicit about what is stable, what is nightly-only, and what is under active discussion. Rust continues to evolve; this book targets the language as it exists now while flagging features that may change your mental models in the near future.

Finally, remember that expert Rust development is not about using every advanced feature available. It is about choosing the right tool for each problem, understanding its costs and benefits, and building systems that are correct, efficient, and maintainable over time. The goal of this book is to give you the knowledge and mental models to make those choices with confidence.

Let us begin.

Chapter 1: Advanced Ownership and Borrowing

Ownership is the foundation of Rust, but most developers stop at understanding its surface rules: one owner at a time, references obey borrowing constraints, values drop when they go out of scope. Expert-level Rust requires going deeper — understanding how the borrow checker actually works as a constraint solver, how variance affects lifetime relationships, how destructors compose across nested types, and how to reason about ownership in large codebases where simple patterns are insufficient.

This chapter covers those advanced topics. We will examine the borrow checker's internal logic, explore variance in detail, understand drop order guarantees, tackle complex ownership patterns across module boundaries, preview Pinning for self-referential structures, and identify subtle pitfalls that can cause use-after-free even in code that looks safe.

The Borrow Checker as a Constraint Solver

When you write Rust code, the borrow checker does not simply scan for violations of "one mutable reference or many immutable references." It performs a sophisticated analysis on your program's Mid-Level Intermediate Representation (MIR) to construct and solve a system of constraints about when borrows are alive and what operations occur while they exist.

The modern borrow checker uses Non-Lexical Lifetimes (NLL), which means borrow scopes are determined by actual usage, not by lexical block boundaries. Consider this code:

```
fn example() -> &'static str {
    let s = String::from("hello");
    let r = &s;
    // s is no longer used here, so its borrow ends at the last use of `r`
    // But we still cannot return `r` because `s` drops at end of function
    r
}
```

With NLL, the mutable or immutable borrow of s extends only as long as necessary — from where it is created until its last actual use. This allows patterns that would be rejected under older lexical lifetime rules:

```
fn demonstrate_nll() {
    let mut data = vec![1, 2, 3];
    {
        let first = &mut data[0];
        *first = 100;
        // `first` is no longer used after this point
        // NLL allows a new borrow here because the mutable borrow ended at
↪   last use of `first`
    }
    let second = &data[1];
    println!("second: {}", second);
}
```

Under NLL, the mutable borrow through first ends when first is last used (at the assignment), not at the closing brace. This enables more natural code patterns without sacrificing safety.

The borrow checker's analysis proceeds in phases. First, it converts your HIR (High-Level Intermediate Representation) into MIR, a three-address-code form where every operation is explicit and control flow is clear. Then it performs borrowck on the MIR:

1. It identifies all borrow sites — places where references are created
2. For each borrow, it tracks where the borrowed place is accessed
3. It checks that at every program point, the aliasing rules hold: if a mutable borrow exists, no other reference to that place can exist; if an immutable borrow exists, no mutable borrow can exist

The key insight for expert developers is that the borrow checker operates on places — memory locations identified by a path through your data structures —

not on variable names. Two references alias if they point to overlapping places, regardless of how those references were created. This is why patterns like this fail:

```
fn cannot_do_this() {
    let mut x = 5;
    let r1 = &x;
    let r2 = &mut x; // ERROR: cannot borrow `x` as mutable because it is also
↪  borrowed as immutable
    println!("{}", r1);
}
```

The place x is borrowed immutably through r1, so creating a mutable borrow through r2 would violate the aliasing rules. The error occurs at the creation of r2, not when both references are used simultaneously.

Understanding this helps you reason about more complex scenarios. Consider a function that takes multiple references to fields of the same struct:

```
struct Point {
    x: f64,
    y: f64,
}

fn move_toward(p: &mut Point, target: &Point) {
    // This works because `p` and `target` are different places
    p.x += (target.x - p.x) * 0.1;
    p.y += (target.y - p.y) * 0.1;
}

fn problem() {
    let mut origin = Point { x: 0.0, y: 0.0 };
    let dest = Point { x: 10.0, y: 10.0 };
    move_toward(&mut origin, &dest); // Fine: different structs
}

fn self_move_problem() {
    let mut p = Point { x: 0.0, y: 10.0 };
    // ERROR: cannot borrow `p` as immutable because it is already borrowed as
↪  mutable
    move_toward(&mut p, &p);
}
```

In self_move_problem, both arguments refer to the same place p. The mutable borrow through the first argument and the immutable borrow through

the second would alias, violating Rust's rules. This is not a limitation of the borrow checker; it is a safety requirement. If move_toward stored target.x somewhere and later mutated p.x, we could observe inconsistent state.

Variance: Covariance, Contravariance, and Invariance

Variance describes how subtyping relationships propagate through type constructors. In Rust, subtyping exists only for lifetimes — a longer lifetime is a subtype of a shorter lifetime because data that lives longer can be used wherever data that lives shorter is expected. Variance determines whether this relationship is preserved, reversed, or blocked when lifetimes appear inside generic types.

There are three variance relationships:

- **Covariant**: preserves the subtyping direction. If 'long <: 'short and F is covariant in its lifetime parameter, then F<'long> <: F<'short>.
- **Contravariant**: reverses the subtyping direction. If 'long <: 'short and F is contravariant, then F<'short> <: F<'long>.
- **Invariant**: blocks subtyping entirely. Even if 'long <: 'short, F<'long> and F<'short> are unrelated types.

These relationships are not arbitrary; they follow from how the type is used. The compiler computes variance automatically based on the positions where lifetime parameters appear.

Immutable references are covariant in their lifetime:

```
fn demonstrate_covariance() {
    let long_lived = String::from("hello");
    let r_long: &String = &long_lived;

    // &'long String can be used wherever &'short String is expected
    let r_short: &String = r_long; // Covariant coercion: OK

    fn takes_short_ref(_r: &String) {}
    takes_short_ref(r_long); // Also OK via covariance
}
```

This is safe because an immutable reference only reads data. If you have a reference valid for a long lifetime, you can safely use it as a reference valid for a shorter lifetime — the data will definitely still exist during that shorter period.

Mutable references are invariant in their lifetime:

```
fn demonstrate_invariance() {
    let long_lived = String::from("hello");
    let r_long: &mut String = &mut long_lived;

    // This would be unsound if allowed:
    // let r_short: &mut String = r_long; // ERROR: cannot coerce due to invariance
}
```

Invariance is necessary because mutable references can both read and write. If we allowed covariant coercion from &'long mut T to &'short mut T, we could create a scenario where the shorter-lived reference outlives its expected scope while still pointing to data that might have been modified through the longer-lived reference. More concretely, invariance prevents use-after-free bugs that arise from mixing borrows at different lifetimes.

The critical type that forces invariance is UnsafeCell:

```
use std::cell::UnsafeCell;

struct InvariantExample<'a> {
    data: UnsafeCell<&'a i32>,
}

// InvariantExample<'a> is invariant in 'a because UnsafeCell<T> is invariant in T.
// This prevents unsound lifetime coercion through interior mutability.
```

UnsafeCell enables interior mutability — mutation through a shared reference. If types containing UnsafeCell were covariant, we could exploit that to mutate data through a reference with an incorrect lifetime. The compiler therefore marks any type containing UnsafeCell as invariant.

Functions exhibit contravariance in their argument types:

```
fn demonstrate_contravariance() {
    // A function that accepts a shorter-lived reference can be used
    // wherever a function accepting a longer-lived reference is expected.

    fn takes_short<'a>(x: &'a i32) -> i32 { *x }
    fn takes_long<'a>(x: &'a i32) -> i32 { *x }

    // Conceptually: fn(&'short i32) is "more general" than fn(&'long i32)
    // because it can accept references with any lifetime, including short ones.
    // This is contravariance in action.
}
```

Contravariance for functions makes intuitive sense once you think about it: if you need a function that accepts long-lived data, and someone offers you a function that accepts short-lived data, that function can definitely handle your long-lived data (because long-lived data satisfies the shorter lifetime requirement). The acceptance direction is reversed.

Understanding variance matters in practice when designing generic APIs. Consider this subtle bug:

```
use std::cell::RefCell;

struct Container<'a> {
    value: RefCell<&'a str>,
}

impl<'a> Container<'a> {
    fn new(value: &'a str) -> Self {
        Container { value: RefCell::new(value) }
    }

    fn get(&self) -> &str {
        *self.value.borrow()
    }
}

fn variance_bug() {
    let long = String::from("long-lived");
    let container: Container<'static>;

    {
        let short = String::from("short-lived");
        let temp = Container::new(&short);
        // If Container were covariant in its lifetime, we could coerce here:
        // container = temp; // This would be unsound!
        // `temp` borrows `short`, but `container` claims 'static lifetime.
```

```
        // When `short` drops, `container.value` becomes a dangling reference.
    }

    // RefCell is invariant because it contains UnsafeCell, so this coercion is
    ↪ blocked.
    // The compiler prevents the use-after-free at compile time.
}
```

RefCell's invariance saves us here. If Container were covariant, we could smuggle a short-lived borrow into a long-lived container and create a dangling reference. Variance is not an academic curiosity; it is a safety mechanism that prevents real bugs.

Drop Semantics and Destructor Order

Rust's deterministic destruction is one of its most powerful features for resource management. Unlike garbage-collected languages where finalization order is undefined, Rust guarantees exactly when destructors run and in what order. Understanding these guarantees is essential for writing correct code that manages multiple resources with dependencies between them.

The Drop trait defines a destructor:

```
struct TrackedResource {
    name: String,
}

impl Drop for TrackedResource {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}

fn main() {
    let a = TrackedResource { name: "a".into() };
    let b = TrackedResource { name: "b".into() };
    // `b` drops first, then `a`: reverse order of initialization
}
// Output: Dropping b
//         Dropping a
```

For local variables in a function, drop order is deterministic and follows reverse initialization order. The last variable initialized is the first to drop. This is guaranteed by the language specification.

For struct fields, drop order follows declaration order – not reverse declaration order as in C++. This is a deliberate design choice that Rust has stabilized:

```
struct MultiResource {
    database: TrackedResource,
    cache: TrackedResource,
    logger: TrackedResource,
}

fn main() {
    let mr = MultiResource {
        database: TrackedResource { name: "database".into() },
        cache: TrackedResource { name: "cache".into() },
        logger: TrackedResource { name: "logger".into() },
    };
}
// Output: Dropping database
//         Dropping cache
//         Dropping logger
```

Fields drop in declaration order. This means if your cache depends on the database being alive, declare database after cache so it drops last. If your logger should outlive everything else for final log writes, declare it last.

When a type implements Drop explicitly, its custom drop logic runs before field drops:

```
struct WithCustomDrop {
    inner: TrackedResource,
}

impl Drop for WithCustomDrop {
    fn drop(&mut self) {
        println!("Custom drop running first");
        // `inner` will be dropped automatically after this function returns
    }
}

fn main() {
    let w = WithCustomDrop {
        inner: TrackedResource { name: "inner".into() },
    };
}
// Output: Custom drop running first
//         Dropping inner
```

This ordering matters when your Drop implementation needs to use other fields. They are still alive and accessible during your custom drop logic.

There is a subtle interaction between Drop and ownership that can cause double-free bugs in unsafe code. Consider this incorrect pattern:

```
use std::alloc::{dealloc, Layout};

struct BadWrapper {
    ptr: *mut u8,
    layout: Layout,
}

impl BadWrapper {
    fn new(size: usize) -> Self {
        let ptr = unsafe { std::alloc::alloc(Layout::from_size_align(size, 1).unwrap()) };
        BadWrapper { ptr, layout: Layout::from_size_align(size, 1).unwrap() }
    }
}

impl Drop for BadWrapper {
    fn drop(&mut self) {
        // BUG: We manually deallocate, but then Rust will also try to drop `self.ptr`
        // Since *mut u8 is Copy and has no Drop impl, this particular case works,
        // but if ptr were a Box<T>, we'd double-free.
        unsafe {
            dealloc(self.ptr, self.layout);
        }
    }
}
```

The bug here is more subtle than it appears. If BadWrapper contained a Box instead of a raw pointer, our manual dealloc would free the memory, and then Rust's automatic drop glue would call Box::drop on the same pointer — double-free, undefined behavior.

The correct pattern uses ManuallyDrop or Option to prevent automatic dropping:

```rust
use std::alloc::{dealloc, Layout};
use std::mem::ManuallyDrop;

struct GoodWrapper {
    ptr: *mut u8,
    layout: Layout,
}

impl GoodWrapper {
    fn new(size: usize) -> Self {
        let ptr = unsafe { std::alloc::alloc(Layout::from_size_align(size, 1).unwrap()) };
        GoodWrapper { ptr, layout: Layout::from_size_align(size, 1).unwrap() }
    }
}

impl Drop for GoodWrapper {
    fn drop(&mut self) {
        unsafe {
            dealloc(self.ptr, self.layout);
        }
        // Raw pointers have no Drop impl, so nothing else happens here. Safe.
    }
}

struct WrapperWithBox {
    inner: ManuallyDrop<Box<[u8]>>,
}

impl Drop for WrapperWithBox {
    fn drop(&mut self) {
        // We manage the Box manually; ManuallyDrop prevents automatic Drop.
        let boxed = unsafe { ManuallyDrop::take(&mut self.inner) };
        let ptr = Box::into_raw(boxed);
        // ... custom cleanup using raw pointer ...
        unsafe {
            dealloc(ptr as *mut u8, std::alloc::Layout::from_size_align(10, 1).unwrap());
        }
    }
}
```

ManuallyDrop wraps a value of type T and suppresses its destructor. You are responsible for manually dropping the value when appropriate. This is the standard tool for customizing drop behavior while maintaining safety.

Complex Ownership Patterns Across Modules

In real-world Rust codebases, ownership patterns become complex as data flows across module boundaries, through trait objects, into concurrent contexts, and back again. Expert developers need strategies for managing these patterns without sacrificing clarity or safety.

One common pattern is the owned-with-borrowed-fields problem: a struct that owns some data but also holds references to external data. This requires careful lifetime management:

```
mod config {
    pub struct Config {
        pub name: String,
        pub timeout: u64,
    }
}

mod service {
    use super::config::Config;

    // Service owns its buffer but borrows configuration
    pub struct Service<'a> {
        config: &'a Config,
        buffer: Vec<u8>,
    }

    impl<'a> Service<'a> {
        pub fn new(config: &'a Config) -> Self {
            Service {
                config,
                buffer: vec![0; config.timeout as usize],
            }
        }

        pub fn process(&mut self, input: &[u8]) {
            self.buffer.extend_from_slice(input);
        }

        pub fn timeout(&self) -> u64 {
            self.config.timeout
        }
    }
}

fn main() {
    let config = config::Config {
```

```
        name: "my-service".into(),
        timeout: 1024,
    };
    let mut service = service::Service::new(&config);
    service.process(b"hello");
    println!("timeout: {}", service.timeout());
}
```

The lifetime 'a ties Service to the Config it borrows. If config were dropped before service, using service would be a use-after-free — but the borrow checker prevents this at compile time.

When lifetimes become too constraining, the newtype pattern or owned wrappers can help:

```
// Instead of borrowing Config, own a copy of what we need
pub struct ServiceOwned {
    timeout: u64,
    buffer: Vec<u8>,
}

impl ServiceOwned {
    pub fn new(config: &Config) -> Self {
        ServiceOwned {
            timeout: config.timeout,
            buffer: vec![0; config.timeout as usize],
        }
    }
}
```

This trades memory (storing a copy of timeout instead of borrowing it) for flexibility (ServiceOwned is not tied to Config's lifetime). For small, cheaply-copyable data, this is often the right trade.

Another complex pattern arises when you need to store heterogeneous types in collections. Trait objects solve this through dynamic dispatch:

```rust
trait Handler {
    fn handle(&self, input: &str);
}

struct LogHandler;
impl Handler for LogHandler {
    fn handle(&self, input: &str) {
        println!("LOG: {}", input);
    }
}

struct MetricsHandler;
impl Handler for MetricsHandler {
    fn handle(&self, input: &str) {
        println!("METRIC: {}", input);
    }
}

fn main() {
    let handlers: Vec<Box<dyn Handler>> = vec![
        Box::new(LogHandler),
        Box::new(MetricsHandler),
    ];

    for handler in &handlers {
        handler.handle("test");
    }
}
```

Each Box is a fat pointer containing the data pointer and a vtable pointer. The vtable stores function pointers for each trait method plus metadata about the concrete type (size, alignment, destructor). Dynamic dispatch incurs an indirection cost — we load the vtable pointer, then load the method pointer from the vtable, then call it. For most applications this overhead is negligible compared to I/O costs, but in tight loops it can matter.

Pinning and Self-Referential Structures (Preview)

Rust's ownership model normally prevents self-referential structures — structs that contain references to their own fields. The problem is straightforward: if you move a struct containing a self-reference, the reference becomes dangling because it still points to the old memory location.

```
// This does not compile:
struct SelfReferential {
    data: String,
    // slice refers into `data`, but we cannot express that lifetime
    ↪ relationship
    slice: &'a str,
}
```

The compiler rejects this because it cannot guarantee that slice will remain valid if SelfReferential is moved. Yet self-referential structures are genuinely useful — parsers that track positions within their input buffer, async state machines that borrow across await points, caches that hold references into owned data.

Rust solves this problem with Pin, introduced in RFC 2349 and stabilized in Rust 1.33. Pin is a wrapper type that guarantees its contents will not be moved in memory after pinning:

```
use std::pin::Pin;

struct PinnedSelfRef {
    data: String,
    slice: *const str, // Raw pointer to bypass borrow checker initially
}

impl PinnedSelfRef {
    fn new(s: String) -> Pin<Box<Self>> {
        let mut boxed = Box::pin(PinnedSelfRef {
            data: s,
            slice: std::ptr::null(),
        });

        // Now that we're pinned (won't move), set the self-reference
        unsafe {
            let this = Pin::get_unchecked_mut(&mut boxed);
            this.slice = this.data.as_str() as *const str;
        }
        boxed
    }

    fn get_slice(self: Pin<&Self>) -> &str {
       // SAFETY: We only set slice after pinning, and pinned values don't move.
        unsafe { &*self.slice }
    }
}

fn main() {
```

```
    let pinned = PinnedSelfRef::new("hello world".to_string());
    println!("slice: {}", pinned.get_slice());
}
```

Pin<Box> creates a heap-allocated, immovable value. Once pinned, the only way to get mutable access is through Pin<&mut T>, which prevents moving the value out. The Unpin marker trait indicates that a type is safe to move even when pinned — most types implement Unpin automatically. Types that do not implement Unpin (like self-referential structs and async futures) require Pin to guarantee immobility.

We will cover Pin in depth in Chapter 8 when discussing async Rust, where it becomes essential for understanding how futures work internally. For now, understand that Pin exists to enable self-referential types safely by trading away movability for correctness.

Pitfalls: Use-After-Free in Safe Code and How to Avoid It

Rust's safety guarantees are strong but not magic. Certain patterns can lead to use-after-free bugs even when no unsafe blocks appear in your code. These bugs typically arise from misunderstandings of how lifetimes, interior mutability, or async cancellation interact.

One common pitfall involves cached references:

```
use std::collections::HashMap;

struct Cache {
    data: HashMap<String, String>,
}

impl Cache {
    // DANGER: Returns a reference into internal storage
    // If cache.insert() triggers a reallocation, this reference dangles!
    fn get_mut(&mut self, key: &str) -> Option<&str> {
        self.data.get(key).map(|s| s.as_str())
    }

    fn insert(&mut self, key: String, value: String) {
        self.data.insert(key, value);
    }
}

```

```
fn cache_bug() {
    let mut cache = Cache { data: HashMap::new() };
    cache.insert("key".into(), "value".into());

    let reference = cache.get_mut("key").unwrap();
    // `reference` points into cache.data's internal storage

    // Inserting many items might cause HashMap to reallocate
    for i in 0..1000 {
        cache.insert(format!("key{}", i), format!("value{}", i));
    }

    // USE-AFTER-FREE: `reference` may now be dangling!
    println!("{}", reference);
}
```

This code has no unsafe blocks but exhibits use-after-free behavior. The HashMap reallocation invalidates the earlier reference. The borrow checker allows this because from its perspective, each call to get_mut and insert is independent — it does not track that the returned reference outlives the function call.

The fix is to avoid returning references into internal storage when mutation might invalidate them:

```
impl Cache {
    // Safe: returns owned data or a guard that prevents mutation
    fn get(&self, key: &str) -> Option<&String> {
        self.data.get(key)
    }
}
```

Another pitfall involves RefCell and panics:

```rust
use std::cell::RefCell;

thread_local! {
    static CONTEXT: RefCell<Option<String>> = RefCell::new(None);
}

fn set_context(s: String) {
    CONTEXT.with(|c| *c.borrow_mut() = Some(s));
}

fn get_context() -> Option<String> {
    CONTEXT.with(|c| c.borrow().clone())
}

fn panic_in_borrow() {
    CONTEXT.with(|c| {
        let _guard = c.borrow(); // Immutable borrow active
        panic!("Oops!"); // Panic unwinds while borrow is held
    });
    // After resume, RefCell may be in a poisoned/broken state
}
```

When a panic occurs while a RefCell borrow is active, the borrow count can become inconsistent. Rust 1.69+ improved this with poison handling, but it remains a hazard. Best practice: keep borrows short and avoid panicking while holding RefCell guards.

Understanding these pitfalls requires thinking about what invariants your code maintain and where those invariants might be violated by reallocation, panic unwinding, or async cancellation. The borrow checker catches many errors, but not all — expert developers supplement it with careful reasoning about runtime behavior.

Chapter 2: The Trait System Deep Dive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Higher-Ranked Trait Bounds (HRTBs) in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Associated Types Versus Generic Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Generic Associated Types (GATs): Problems They Solve

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Trait Objects and Dynamic Dispatch: When and How

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Orphan Rule, Sealed Traits, and API Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Supertraits, Negative Impls, and Trait Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Coherence Failures and Ambiguous Resolutions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Chapter 3: Type-Level Programming and Advanced Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Type-State Pattern: Encoding State in Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Zero-Sized Types: PhantomData and Marker Traits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Dynamically Sized Types (DSTs) Deep Dive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Newtype Pattern: Safety, ABI, and Ergonomics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Compile-Time Computation with Const Generics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Type Inference Limits and Error Explosion

Chapter 4: Interior Mutability and Smart Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Philosophy of Interior Mutability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

RefCell, Cell, and UnsafeCell: The Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Reference Counting: Rc, Arc, Weak, and Cycles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Building Custom Smart Pointers Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Synchronization Primitives as Interior Mutability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Panics in Drop, Deadlocks, and Memory Leaks

Chapter 5: Memory Layout and Representation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Alignment, Padding, and Struct Layout Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

#[repr] Attributes: C, Pack, Transparent, and Integers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Enum Memory Representation and Tagged Unions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Fat Pointers and Metadata

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Cache-Friendly Data Layout and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Undefined Behavior from Misaligned Access

Chapter 6: Concurrency Foundations — Send, Sync, and Atomics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Send and Sync: Auto Traits Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Atomic Operations and Memory Ordering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Sequential Consistency Default and When to Relax It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Building Thread-Safe Abstractions with Auto Traits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Crossbeam and Advanced Concurrency Utilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Data Races in Safe-Looking Code

Chapter 7: Advanced Concurrency and Lock-Free Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Lock-Free Versus Wait-Free: Definitions and Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Implementing a Lock-Free Stack in Rust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Work-Stealing Queues and Parallel Task Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Epoch-Based Reclamation (mimalloc, tikv-jemalloc patterns)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Rayon, Scope-Based Parallelism, and Data Parallels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: ABA Problem, False Sharing, and Priority Inversion

Chapter 8: Asynchronous Rust — Futures, Pinning, and Executors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Future Trait: Polling Semantics from First Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pin and Why Self-Referential Futures Need It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Executors: Work-Stealing, I/O, and Task Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Task Cancellation, JoinHandles, and Scope Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

async/await Desugaring and Common Gotchas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Blocking the Executor, Unpin Bounds, and Pin Projection

Chapter 9: Macros and Metaprogramming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Declarative Macros: Pattern Matching in macro_rules!

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Hygiene, Span Preservation, and Debugging Macros

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Procedural Derive Macros: Generating Boilerplate

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Attribute and Function-Like Procedural Macros

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Token Streams, Parsing with Syn, and Quoting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Macro Explosion, Nightly Instability, and IDE Support

Chapter 10: Unsafe Rust and Safe Abstractions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Contract of unsafe: What You Pledge to Uphold

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Raw Pointers: *const, *mut, Nullability, and Dangling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Aliasing Rules and the Strict Alias Rule in Rust

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Validity Requirements: When Is a Value “Valid”?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Building Safe Abstractions Over Unsafe Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Undefined Behavior That Compiles Cleanly

Chapter 11: FFI and Interoperability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

extern “C” Blocks and Calling Conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Mapping C Types to Rust: Pointers, Structs, and Unions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Linking: Static, Dynamic, and Build Scripts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Callbacks and Function Pointers Across the Boundary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Handling Errors and String Encoding in FFI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: ABI Mismatches, Dangling Pointers, and Lifetime Lies

Chapter 12: Custom Allocators and no_std Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

The Global Allocator Trait and Allocation Internals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Arena Allocators and Bump Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Slab Allocators for Fixed-Size Objects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

no_std Essentials: Core, Alloc, and Panic Handlers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Embedded Rust Patterns: HALs, RTIC, and Cortex-M

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Pitfalls: Stack Overflow, Allocation Failure, and Missing Panics

Chapter 13: Performance Engineering and Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Benchmarking with criterion: Statistical Rigor Over Wall-Clock Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Profiling: From Sampling to Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Build Configuration: Optimization Levels, LTO, and Codegen Units

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Profile-Guided Optimization (PGO)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Cache Locality and Branch Prediction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Monomorphization Costs and Mitigation Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Static Versus Dynamic Dispatch Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Zero-Cost Abstractions in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Chapter 14: Ecosystem, Architecture, and Production Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Error Handling Architecture: anyhow for Applications, thiserror for Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Library Design: API Stability, Sealed Traits, and the Orphan Rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Testing Strategies: Property-Based Testing and Fuzzing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Documentation Practices for Rust Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Advanced Cargo: Workspaces, Features, and Conditional Compilation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Production Architectural Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

Conclusion: The Path to Rust Mastery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/advancedrustprogramming.